Python For Data Science *Cheat Sheet*

PySpark Basics

Learn Python for data science Interactively at www.DataCamp.com

Spark

PySpark is the Spark Python API that exposes the Spark programming model to Python

Initializing Spark

SparkContext

```
>>> from pyspark import SparkContext
>>> sc = SparkContext(master = 'local[2]')
```

Inspect SparkContext

```
>>> sc.version                  Retrieve SparkContext version
>>> sc.pythonVer                Retrieve Python version
>>> sc.master                   Master URL to connect to
>>> str(sc.sparkHome)           Path where Spark is installed on worker nodes
>>> str(sc.sparkUser())         Retrieve name of the Spark User running SparkContext
>>> sc.appName                  Return application name
>>> sc.applicationId            Retrieve application ID
>>> sc.defaultParallelism       Return default level of parallelism
>>> sc.defaultMinPartitions     Default minimum number of partitions for RDDs
```

Configuration

```
>>> from pyspark import SparkConf, SparkContext
>>> conf = (SparkConf()
            .setMaster("local")
            .setAppName("My app")
            .set("spark.executor.memory", "1g"))
>>> sc = SparkContext(conf = conf)
```

Using The Shell

In the PySpark shell, a special interpreter-aware SparkContext is already created in the variable called `sc`.

```
$ ./bin/spark-shell --master local[2]
$ ./bin/pyspark --master local[4] --py-files code.py
```

Set which master the context connects to with the `--master` argument, and add Python .zip, .egg or .py files to the runtime path by passing a comma-separated list to `--py-files`.

Loading Data

Parallelized Collections

```
>>> rdd = sc.parallelize([('a',7),('a',2),('b',2)])
>>> rdd2 = sc.parallelize([('a',2),('d',1),('b',1)])
>>> rdd3 = sc.parallelize(range(100))
>>> rdd4 = sc.parallelize([("a",["x","y","z"]),
                           ("b",["p", "r"])])
```

External Data

Read either one text file from HDFS, a local file system or or any Hadoop-supported file system URI with `textFile()`, or read in a directory of text files with `wholeTextFiles()`.

```
>>> textFile = sc.textFile("/my/directory/*.txt")
>>> textFile2 = sc.wholeTextFiles("/my/directory/")
```

Retrieving RDD Information

Basic Information

```
>>> rdd.getNumPartitions()      List the number of partitions
>>> rdd.count()                 Count RDD instances
 3
>>> rdd.countByKey()            Count RDD instances by key
 defaultdict(<type 'int'>,{'a':2,'b':1})
>>> rdd.countByValue()          Count RDD instances by value
 defaultdict(<type 'int'>,{('b',2):1,('a',2):1,('a',7):1})
>>> rdd.collectAsMap()          Return (key,value) pairs as a dictionary
 {'a': 2,'b': 2}
>>> rdd3.sum()                  Sum of RDD elements
 4950
>>> sc.parallelize([]).isEmpty()  Check whether RDD is empty
 True
```

Summary

```
>>> rdd3.max()                  Maximum value of RDD elements
  99
>>> rdd3.min()                  Minimum value of RDD elements
  0
>>> rdd3.mean()                 Mean value of RDD elements
  49.5
>>> rdd3.stdev()                Standard deviation of RDD elements
  28.866070047722118
>>> rdd3.variance()             Compute variance of RDD elements
  833.25
>>> rdd3.histogram(3)           Compute histogram by bins
  ([0,33,66,99],[33,33,34])
>>> rdd3.stats()                Summary statistics (count, mean, stdev, max & min)
```

Applying Functions

```
>>> rdd.map(lambda x: x+(x[1],x[0]))      Apply a function to each RDD element
       .collect()
  [('a',7,7,'a'),('a',2,2,'a'),('b',2,2,'b')]
>>> rdd5 = rdd.flatMap(lambda x: x+(x[1],x[0]))   Apply a function to each RDD element and flatten the result
>>> rdd5.collect()
  ['a',7,7,'a','a',2,2,'a','b',2,2,'b']
>>> rdd4.flatMapValues(lambda x: x)       Apply a flatMap function to each (key,value) pair of rdd4 without changing the keys
       .collect()
  [('a','x'),('a','y'),('a','z'),('b','p'),('b','r')]
```

Selecting Data

Getting

```
>>> rdd.collect()               Return a list with all RDD elements
  [('a', 7), ('a', 2), ('b', 2)]
>>> rdd.take(2)                 Take first 2 RDD elements
  [('a', 7), ('a', 2)]
>>> rdd.first()                 Take first RDD element
  ('a', 7)
>>> rdd.top(2)                  Take top 2 RDD elements
  [('b', 2), ('a', 7)]
```

Sampling

```
>>> rdd3.sample(False, 0.15, 81).collect()   Return sampled subset of rdd3
  [3,4,27,31,40,41,42,43,60,76,79,80,86,97]
```

Filtering

```
>>> rdd.filter(lambda x: "a" in x)   Filter the RDD
       .collect()
  [('a',7),('a',2)]
>>> rdd5.distinct().collect()        Return distinct RDD values
  ['a',2,'b',7]
>>> rdd.keys().collect()             Return (key,value) RDD's keys
  ['a', 'a', 'b']
```

Iterating

```
>>> def g(x): print(x)
>>> rdd.foreach(g)                   Apply a function to all RDD elements
  ('a', 7)
  ('b', 2)
  ('a', 2)
```

Reshaping Data

Reducing

```
>>> rdd.reduceByKey(lambda x,y : x+y)   Merge the rdd values for each key
       .collect()
  [('a',9),('b',2)]
>>> rdd.reduce(lambda a, b: a + b)      Merge the rdd values
  ('a',7,'a',2,'b',2)
```

Grouping by

```
>>> rdd3.groupBy(lambda x: x % 2)       Return RDD of grouped values
        .mapValues(list)
        .collect()
>>> rdd.groupByKey()                    Group rdd by key
       .mapValues(list)
       .collect()
  [('a',[7,2]),('b',[2])]
```

Aggregating

```
>>> seqOp = (lambda x,y: (x[0]+y,x[1]+1))
>>> combOp = (lambda x,y:(x[0]+y[0],x[1]+y[1]))
>>> rdd3.aggregate((0,0),seqOp,combOp)      Aggregate RDD elements of each partition and then the results
  (4950,100)
>>> rdd.aggregateByKey((0,0),seqop,combop)  Aggregate values of each RDD key
       .collect()
  [('a',(9,2)), ('b',(2,1))]
>>> rdd3.fold(0,add)                        Aggregate the elements of each partition, and then the results
  4950
>>> rdd.foldByKey(0, add)                   Merge the values for each key
       .collect()
  [('a',9),('b',2)]
>>> rdd3.keyBy(lambda x: x+x)               Create tuples of RDD elements by applying a function
       .collect()
```

Mathematical Operations

```
>>> rdd.subtract(rdd2)          Return each rdd value not contained in rdd2
       .collect()
  [('b',2),('a',7)]
>>> rdd2.subtractByKey(rdd)     Return each (key,value) pair of rdd2 with no matching key in rdd
       .collect()
  [('d', 1)]
>>> rdd.cartesian(rdd2).collect()   Return the Cartesian product of rdd and rdd2
```

Sort

```
>>> rdd2.sortBy(lambda x: x[1])     Sort RDD by given function
       .collect()
  [('d',1),('b',1),('a',2)]
>>> rdd2.sortByKey()                Sort (key, value) RDD by key
       .collect()
  [('a',2),('b',1),('d',1)]
```

Repartitioning

```
>>> rdd.repartition(4)          New RDD with 4 partitions
>>> rdd.coalesce(1)             Decrease the number of partitions in the RDD to 1
```

Saving

```
>>> rdd.saveAsTextFile("rdd.txt")
>>> rdd.saveAsHadoopFile("hdfs://namenodehost/parent/child",
                         'org.apache.hadoop.mapred.TextOutputFormat')
```

Stopping SparkContext

```
>>> sc.stop()
```

Execution

```
$ ./bin/spark-submit examples/src/main/python/pi.py
```